Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 797–815
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

# A priority-based distributed group mutual exclusion algorithm when group access is non-uniform ☆

Neeraj Mittal [a,*], Prajwal K. Mohan [b,1]

[a] *Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA*
[b] *Digital Home Group, Intel Corporation, Hillsboro, OR 97124, USA*

Received 5 May 2006; received in revised form 2 January 2007; accepted 19 February 2007
Available online 14 March 2007

**Abstract**

In the group mutual exclusion problem, each critical section has a *type* or a *group* associated with it. Processes requesting critical sections belonging to the same group (that is, of the same type) may execute their critical sections concurrently. However, processes requesting critical sections belonging to different groups (that is, of different types) must execute their critical sections in a mutually exclusive manner.

Most algorithms for solving the group mutual exclusion problem that have been proposed so far in the literature treat all groups equally. This is quite acceptable if a process, at the time of making a request for critical section, selects a group for the critical section uniformly. However, if some groups are more likely to be selected than others, then better performance can be achieved by treating different groups in a different manner.

In this paper, we propose an efficient algorithm for solving the group mutual exclusion problem when group selection probabilities are non-uniformly distributed. Our algorithm has a message complexity of $2n - 1$ per request for critical section, where $n$ is the number of processes in the system. It has low synchronization delay of one message hop and low waiting time of two message hops. The maximum concurrency of our algorithm is $n$, which implies that if all processes have requested critical sections of the same type then all of them may execute their critical sections concurrently. Finally, the *amortized* message-size complexity of our algorithm is $O(1)$.

Our experimental results indicate that our algorithm *outperforms* the existing algorithms, whose complexity measures are comparable to that of ours, by as much as 50% in some cases.
© 2007 Elsevier Inc. All rights reserved.

*Keywords:* Message-passing system; Resource management; Group mutual exclusion; Token-based algorithm; Non-uniform group selection; Priority-based scheme

## 1. Introduction

Resource management is one of the most important and fundamental problems in distributed systems. In many cases, to maintain the integrity of a resource, at most one process should access the resource at any time. As a result, accesses to the same resource (that is, execution of critical sections) by different processes have to be serialized. This problem is referred to as the *mutual exclusion problem*. Mutual exclusion has been studied extensively and a large number of solutions have been developed under both shared-memory model (*e.g.*, [9,15]) and message-passing model (*e.g.*, [24,17,27,23]). Many different variants of mutual exclusion have also been defined. Some examples of variants include *k-mutual exclusion problem* [11], *dining philosophers problem* [10], *drinking philosophers problem* [6] and *committee coordination problem* [7].

Recently, Joung has proposed another variant of the mutual exclusion problem called the *group mutual exclusion problem* [12]. In the group mutual exclusion (GME) problem, every critical section is associated with a *type* or a *group*. Critical sections belonging to the same group can be executed concurrently. However critical sections belonging to different groups must be executed in a mutually exclusive manner. Intuitively,

☆ A preliminary version of this paper has appeared earlier in 2005 Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS) [20].

* Corresponding author. Fax: +1 972 883 2349.
*E-mail addresses:* neerajm@utdallas.edu (N. Mittal), prajwal.karur.mohan@intel.com (P.K. Mohan).

[1] This work was done when the author was a student in the Department of Computer Science at the University of Texas at Dallas.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.02.005

if two critical sections belong to the same group, then their requesting processes share some common property. The multiple-readers/single-writer (MRSW) problem is a special case of the group mutual exclusion problem. Let $n$ denote the number of processes in the system. The multiple-readers/single-writer problem can be modeled using $n + 1$ groups. All read critical sections belong to the same group. On the other hand, write critical sections requested by different processes belong to different groups. (There is one group for each process.) As another application of the problem, assume that data are stored on CDs in a CD-jukebox and only one disk can be loaded for access at any time [12]. Clearly, when a disk is loaded, users who need data on the currently loaded disk can access the disk concurrently. On the other hand, users who need access to a different disk have to wait until the currently loaded disk is unloaded. Manabe and Park [18] have proposed an extension to group mutual exclusion in which, at the time of request, a process is allowed to specify more than one type; the request can be fulfilled as long as the process gets to execute a critical section of any one of those types. This corresponds to the case when the same data are replicated on multiple CDs and any of those CDs can be used to satisfy a request for that data.

Most algorithms for solving the group mutual exclusion problem that have been developed so far treat all groups in the same manner [12,29,4,3,14,2]. This is quite acceptable, if a process, at the time of making a request for critical section, selects the group for the critical section uniformly. However, depending on the application, some groups may be more likely to be selected (or requested) than others. For instance, in the CD jukebox example, some CDs may contain data that are accessed more frequently than data on other CDs. As another example, in the MRSW problem, read requests may be more common than write requests. If group selection probability is non-uniformly distributed, then better performance can be achieved by differentiating between various groups. For example, groups with many outstanding requests for critical section may be given priority over groups with only a few outstanding requests for critical section. In this paper, we present an efficiently distributed algorithm for solving the group mutual exclusion problem that is especially suitable for applications in which group selection probability may be non-uniformly distributed.

### 1.1. Related work

In [13], Joung modified Ricart and Agrawala's algorithm for traditional mutual exclusion [24] to derive two algorithms for group mutual exclusion, namely RA1 and RA2. The message complexity of the two algorithms (RA1 and RA2) is $2(n-1)$ messages and $3(n-1)$ messages (amortized over all requests), respectively, where $n$ is the number of processes in the system. Their synchronization delay, typically measured when the system is heavily loaded, is one message hop. Further, their waiting time, typically measured when the system is lightly loaded, is two message hops. Moreover, both algorithms have maximum concurrency of $n$, which implies that if all processes have requested critical sections of the same type, then it is possible for all of them to execute their critical sections concurrently. However, RA1 has low expected concurrency of $O(1)$ under heavy loads, whereas RA2 has high message-size complexity of $O(n)$. (A message in RA2 may have to carry a vector timestamp of size $n$.)

In [14], Joung proposed two quorum-based algorithms for group mutual exclusion, namely Maekawa_M and Maekawa_S, both of which are derived from Maekawa's quorum-based algorithm for traditional mutual exclusion [17]. Joung also proposed a quorum system suitable for group mutual exclusion called the *surficial quorum system* [14]. The first algorithm Maekawa_M has high message complexity of $O(\frac{nq}{d})$, where $q$ is the maximum size of a quorum and $d$ is the degree of the group quorum system. However, it preserves the synchronization delay and waiting time of the original Maekawa's algorithm. The second algorithm Maekawa_S has message complexity of only $O(q)$, where $q$ is the maximum size of a quorum. However, the reduction in message complexity is achieved at the expense of increased synchronization delay and waiting time, which are both $O(q)$ message hops.

Toyomura et al. [28] presented a quorum-based algorithm for group mutual exclusion that is similar to Maekawa_M. However, unlike Maekawa_M which uses surficial quorum system, the algorithm by Toyomura et al. [28] uses a traditional quorum system. In [2], Atreya and Mittal proposed another quorum-based group mutual exclusion algorithm based on the notion of *surrogate quorum*. Their algorithm has low message complexity of $O(q)$. Moreover, it preserves the synchronization delay and waiting time of the original Maekawa's algorithm. However, the message-size complexity increases to $O(b)$, where $b$ denotes the maximum number of quorums to which a process may belong.

Algorithms for group mutual exclusion problem have also been developed for *ring-based* networks (processes are arranged in a ring) [29,4] and *tree-based* networks (processes are arranged in a tree) [3,16]. However, algorithms for ring- and tree-based networks tend to have high synchronization delay and high waiting time.

### 1.2. Our contribution

In this paper, we devise an efficient token-based algorithm for group mutual exclusion, which is especially suited for applications in which group selection is non-uniform. Our algorithm is derived from Suzuki and Kasami's token-based algorithm for traditional mutual exclusion [27]. Our algorithm has a message complexity of $2n - 1$ messages per request for critical section. It has low synchronization delay of one message hop and low waiting time of two message hops. The maximum concurrency of our algorithm is $n$. Further, our algorithm has high expected concurrency under heavy loads (as in the case of RA2) and low *amortized* message-size complexity of $O(1)$ (as in the case of RA1). Our experimental results indicate

that our algorithm *significantly outperforms* the existing group mutual exclusion algorithms, which have *comparable* complexity measures, by as much as 50% in some cases.

### 1.3. Paper organization

This paper is organized as follows. We present our system model and formally describe the group mutual exclusion problem in Section 2. Our token-based algorithm for group mutual exclusion is discussed in Section 3. We present our experimental results in Section 4. In Section 5, we discuss several enhancements to the basic token-based algorithm to improve its performance and make it tolerant to process crashes. Finally, we present our conclusions and outline future research directions in Section 6.

## 2. Model and problem definition

### 2.1. System model

We assume an asynchronous message-passing distributed system comprising of a set of $n$ processes $P=\{p_1, p_2, \ldots, p_n\}$, which communicate with each other by sending messages over a set of channels. We assume that there is a channel between every pair of processes. Unless otherwise stated, we assume that processes are non-faulty and channels are reliable. We do not assume any global clock or shared memory. Finally, we assume that message delays are finite but may be unbounded.

### 2.2. The group mutual exclusion problem

The problem of *group mutual exclusion* (GME) was first proposed by Joung in [12] as an extension of the traditional mutual exclusion problem. In this problem, every request for a critical section is associated with a *type* or a *group*. Any algorithm for group mutual exclusion should satisfy the following properties:

- *group mutual exclusion (safety)*: at any time, no two processes, which have requested critical sections belonging to different groups, are in their critical sections simultaneously.
- *starvation freedom (liveness)*: a process wishing to enter critical section will succeed eventually.

Intuitively, if two critical sections belong to the same group, then their requesting processes share some common property. As explained earlier, the multiple-readers/single-writer problem is a special case of the group mutual exclusion problem. Clearly, any algorithm for solving the traditional mutual exclusion problem also solves the group mutual exclusion problem. However, such algorithms are sub-optimal because they force all critical sections to be executed in a mutually exclusive manner and therefore do not permit any concurrency whatsoever. To avoid such degenerate solutions and unnecessary synchronization, Joung argued that an algorithm for group mutual exclusion should satisfy the following desirable property:

- *concurrent entry (non-triviality)*: if all requests are for critical sections belonging to the same group, then a requesting process should not be required to wait for entering its critical section until some other process has left its critical section.

Intuitively, the concurrent entry property requires that, when all requests are for critical sections of the same type, then a requesting process should be able to enter its critical section eventually even if none of the processes currently in their critical section ever leave their critical sections. We use $m$ to denote the number of groups in the system. For the multiple-readers/single-write problem, for example, $m = n + 1$.

For convenience, we use the term *session* to refer to a time-interval in which all critical sections executed are of the same type. A session commences with the start of the first critical section and terminates with the end of the last critical section.

#### 2.2.1. Complexity measures

To measure the performance of a group mutual exclusion algorithm, we use the following metrics:

- *message complexity*: the number of messages exchanged per request for critical section;
- *message-size complexity*: the amount of data piggybacked on a message (in terms of number of integers);
- *waiting time*: the time elapsed between when a process issues a request for critical section and when it actually enters the critical section;
- *synchronization delay*: the time elapsed between when the current session terminates and when the next session (of some other type) can commence;
- *system throughput*: the number of critical section requests fulfilled per unit time;
- *concurrency*: the number of processes that are in a session at the same time.

The first five metrics are used to evaluate the performance of a traditional mutual exclusion algorithm as well. The sixth metric is specific to a group mutual exclusion algorithm. We measure synchronization delay and waiting time in terms of number of message hops rather than in terms of time.

Message complexity and message-size complexity together capture the overhead imposed on the communication network by the group mutual exclusion algorithm at run-time. Synchronization delay is measured when the system is heavily loaded and a large number of processes are competing among themselves for accessing the resource. Intuitively, synchronization delay and concurrency measure the system throughput that can be achieved when the system is heavily loaded. The lower the synchronization delay and higher the concurrency, the higher is the system throughput. Waiting time captures the amount of time an application process has to wait for its request to be fulfilled. Waiting time is typically measured when the system is lightly loaded and, therefore, there is no contention for the resource.

## 3. A new token-based algorithm for group mutual exclusion

In this section, we describe our token-based algorithm for group mutual exclusion. Our algorithm is an extension of the Suzuki and Kasami's algorithm for traditional mutual exclusion [27]. Note that any algorithm that solves the group mutual exclusion problem also solves the mutual exclusion problem. Therefore, not surprisingly, all algorithms for group mutual exclusion to our knowledge have been obtained by modifying some mutual exclusion algorithm. For the sake of completeness, we first provide a brief description of the Suzuki and Kasami's token-based algorithm for (traditional) mutual exclusion. We then describe our token-based algorithm for group mutual exclusion, prove its correctness and also analyze its performance.

### *3.1. Background: Suzuki and Kasami's token-based algorithm for traditional mutual exclusion*

Suzuki and Kasami's algorithm [27] achieves mutual exclusion by maintaining a *unique* token in the system. A process can enter the critical section only if it holds the token. The token is said to be busy if its holder is currently in the critical section; otherwise it is *idle*. Every process maintains a vector containing the sequence number of the latest request made by each process that it is aware of. The token contains a first-in-first-out (FIFO) queue of process that have requests waiting to be fulfilled. Additionally, the token also contains a vector that maintains the number of requests that have been fulfilled for each process.

A process, on generating a request for critical section, sends a REQUEST message to other processes if it does not have the token already. The process holding the token, on learning about a pending request, sends the token to the requesting process (via a TOKEN message) as soon as the token becomes idle. The message complexity of Suzuki and Kasami's algorithm is $n$ messages per request for critical section. Its synchronization delay is one message hop and its waiting time is two message hops. The amortized message-size complexity $O(1)$ because, for each request for critical section, there are $n-1$ REQUEST messages of size $O(1)$ and one TOKEN message of size $O(n)$.

### *3.2. The main idea*

If Suzuki and Kasami's algorithm is used for group mutual exclusion without any modification, then all critical sections will be executed one-by-one in a mutually exclusive manner. To enhance concurrency of Suzuki and Kasami's basic algorithm, we use multiple tokens. There are two kinds of tokens: *primary* and *secondary*. At any time, there is exactly one primary token in the system. However, the number of secondary tokens may vary from time-to-time. Initially, process $p_1$ has the primary token, and the number of secondary tokens in the system is zero.

A token has a type (or a group) associated with it and it can only be used to enter critical section of that type (or group). Similar to Suzuki and Kasami's algorithm, a process can enter a critical section of certain type only if it holds a token—primary or secondary—of that type. The difference between a primary and a secondary token is the following: a process holding a primary token is allowed to issue secondary tokens to other processes. However, a process holding a secondary token is not allowed to issue token to any process.

A process, on requesting a critical section, checks whether it has a token of the same type as the critical section requested. If the process is indeed holding such a token, then it can simply enter the critical section without further delay. Otherwise, it sends a REQUEST message to all processes in the system.

Every process $p_i$ maintains a vector $request_i$; the $j$th entry of $request_i$ contains the sequence number of the latest request of process $p_j$, along with its type, that process $p_i$ knows of. It is given by $request_i[j].number$ and $request_i[j].type$. A token—primary as well as secondary—contains a vector *fulfilled*; the $j$th entry of *fulfilled* captures the number of requests of process $p_j$ that have been fulfilled so far as per the token. If a process $p_i$ holds a token, then it can determine whether a process $p_j$ has a pending request by comparing the $j$th entries of $request_i$ and $token_i.fulfilled$: $p_j$ has a pending (or new) request as per $p_i$ if $request_i[j].number > token_i.fulfilled[j]$. The process holding the primary token, on receiving an "unfulfilled" request for critical section of the same type as the type of the token, issues a secondary token to the requesting process. Additionally, a primary token also contains a queue of pending requests. In case a request cannot be fulfilled immediately, which happens if its type conflicts with that of the token, the request is inserted in the token queue, if not already present.

A process holding a token keeps the token until it becomes aware of a *conflicting* pending request, that is, a pending request for critical section of type that is different from the type of the token. Specifically, a process holding a secondary token, on learning about a conflicting pending request, releases the token by sending a RELEASE message to all processes once it is no longer using the token. On the other hand, a process holding the primary token, on learning about a conflicting pending request, passes the primary token to another process with a pending request in the token queue once the token becomes idle. To select the next primary token holder, we first determine the type or group of the next session that should be initiated using a *priority-based scheme* discussed later. Once the type of the next session has been determined, one of the processes that has requested a critical section of the selected type is chosen to become the next primary token holder. All other processes with pending requests for the selected type are issued secondary tokens.

A process $p_i$ on receiving a token from the previous primary token holder $p_j$ may not be able to use the token immediately. In other words, the token may not be *safe* for use. This is because a non-zero number of secondary tokens may have been issued in the previous session and some of these tokens may still be busy. Clearly, process $p_i$ should wait until these tokens have been released before it can use its token to enter the critical

section. To that end, we associate a sequence number with each token that represents the session to which the token belongs (given by variable *session* in the token). The sequence number is incremented whenever the current primary token holder sends the (primary) token to the next primary token holder. A process, on releasing a secondary token, piggybacks this sequence number in the secondary token on the RELEASE message it sends.

We also associate a status with each token indicating whether the token is safe for use, that is, whether all tokens issued in the previous session have been released (given by variable *safe* in the token). Each process records the number of RELEASE messages it has received containing the most recent session sequence number (given by variables $noOfReleaseMessages_i$ and $session_i$ for process $p_i$). Further, a primary token contains the number of secondary tokens that were issued for the previous session (given by variable *oldNoOfTokens* in the token). To determine if a token it has received has become safe for use, process $p_i$ evaluates the following condition:

$$(token_i.oldNoOfTokens = 0) \bigvee \Big((session_i = token_i.session - 1) \wedge (noOfRelease_i = token_i.oldNoOfTokens)\Big). \quad (1)$$

Clearly, if no secondary tokens were issued in the previous session, then the token is safe for use. Otherwise, the token is safe for use if process $p_i$ has received release messages for all secondary tokens issued in the previous session. The former condition is tested by the first disjunct and the latter by the second disjunct. Additionally, the token is also safe for use if process $p_i$ has received a RELEASE message for the current session from some other process, say $p_k$. This implies that process $p_k$ deemed it safe to enter the critical section. This could have happened only if process $p_k$ received all RELEASE messages for the previous session or was told by some other process (via a RELEASE message) that it is safe to use the token. Therefore we modify the condition for testing for the safety of a token as follows:

$$(1) \vee (session_i = token_i.session). \quad (2)$$

In general, there may be a delay between when a process receives a token and when it becomes safe for use. Clearly, a process cannot enter its critical section until the token has become safe. However, the token in some sense is still *busy* during this duration even though its token holder is not executing its critical section. Therefore we consider a token to be busy if the token holder is either (1) currently executing its critical section, or (2) has an outstanding request for critical section and the token is not yet safe for use. Otherwise, we consider it to be idle.

Note that, in our algorithm, the relationship between a primary token holder and a secondary token holder of the same session is *different* from that of a leader and its follower in RA2 [13,2]. Specifically, in our algorithm, it is possible for a secondary token holder to enter the critical section and perhaps even leave it before the primary token holder is able to enter the critical section. This may happen, for example, when a process receives a secondary token from the primary token holder of the previous session before the token arrives at the primary token holder of the current session.

#### *3.2.1. Criterion for selecting the type/group for the next session*

To minimize the waiting time, it is preferable that the next session be of type for which the number of pending requests in the token queue is the *maximum*. However, this simple approach may lead to starvation of a request. To avoid starvation, every pending request in the token queue is associated with an attribute called *age*; it measures the number of sessions that have been initiated since the request was added to the queue. Therefore, every time a new session is initiated, the age of all (pending) requests in the token queue increases by one.

Now, to select the type for the next session, the set of all outstanding requests in the primary token queue is divided into subsets based on request type. All requests for the same type of critical section belong to the same subset. Each type with at least one outstanding request is then assigned a priority, which consists of two parts. The first part depends on the *subset size*, which is given by the number of requests in that subset. The second part depends on the *subset age*, which is given by the sum of the ages of all the requests in that subset. The priority of a type is then computed by simply adding the two parts. The next session that is selected is of type for which the corresponding subset has the *maximum* priority value. As explained earlier, the requesting process for one of the requests in the subset is selected to become the next primary token holder and requesting processes for all other requests in the subset become secondary token holders. We show that this approach prevents starvation, that is, it ensures that every request for critical section is eventually fulfilled.

Note that, when the system is heavily loaded, the type for the next session is selected even before the current session has terminated. It is possible that, when the next session actually starts, some other type may have higher priority than the type that was selected earlier. Therefore our algorithm does not ensure that the type of the session has the highest priority among all types when the session is *initiated*.

### *3.3. Formal description*

A formal description of the algorithm is given in Figs. 1–4. We refer to our algorithm as TokenGME. For convenience, we assume that when a process broadcasts a message to all processes it also sends a copy to itself (which, of course, can be simulated by a local action).

Action A0 initializes the variables of a process. Action A1 is executed when a process generates a request for critical section and action A4 is executed when it leaves the critical section. Actions A2, A3 and A5 are executed on receiving REQUEST, TOKEN and RELEASE messages, respectively. Finally,

Group mutual exclusion algorithm for process $p_i$:

Variables:
$request_i$ : vector $[1..n]$ of tuple $\langle number, type\rangle$;
$session_i$ : sequence number of the latest session that $p_i$ knows of via RELEASE messages;
$noOfReleaseMessages_i$ : number of RELEASE messages that $p_i$ has received for $session_i$;

$token_i$ : token with the following attributes:
$fulfilled$ : vector $[1..n]$ of number of fulfilled requests for each process;
$idle$ : whether the token is in use currently;
$type$ : type of critical section for which the token can be used;
$session$ : sequence number of the session to which the token belongs;
$oldNoOfTokens$ : number of secondary tokens that were issued for the previous session;
$safe$ : have all secondary token holders from the previous session released their tokens?

*// the next two attributes are defined only for a primary token*
$queue$ : priority queue of pending requests;
$noOfTokensIssued$ : number of secondary tokens that have been issued for the current session so far;

*// the next variable is defined only if process $p_i$ has a token*
$isPrimary_i$ : is the token a primary token?

Useful expression:
$safe_i \triangleq (session_i = token_i.session) \bigvee (token_i.oldNoOfTokens = 0) \bigvee ((session_i = token_i.session - 1) \wedge (noOfReleaseMessages_i = token_i.oldNoOfTokens))$

(A0) Initial action: *// initialize all variables*
$\forall k :: request_i[k].number := 0$;
$session_i := 0$;
$noOfReleaseMessages_i := 0$;
**if** $i = 1$ **then**
$\forall k :: token_i.fulfilled[k] := 0$;
$token_i.idle :=$ true;
$token_i.session := 0$;
$token_i.oldNoOfTokens := 0$;
$token_i.queue :=$ empty queue;
$token_i.noOfTokensIssued := 0$;
$isPrimary_i :=$ true;
**else** $token_i :=$ **null** **endif**;

Fig. 1. Token-based group mutual exclusion algorithm.

action A6 is executed whenever the current primary token holder decides to close the current session and start a new session.

### 3.4. Proof of correctness

In this section, we show that TokenGME satisfies safety, liveness and non-triviality properties. Clearly, we ensure that the system contains exactly one primary token at all times. Therefore,

At any time, there is exactly one primary token in the system. (3)

We use a greek letter (*e.g.*, $\alpha$, $\beta$, etc.) to denote a token which may be currently in transit or held by a process. Let $\alpha$ and $\beta$ denote two (possibly same) tokens in the system. Fig. 5 describes the notation used in our proof. New tokens can only be issued by a process holding the primary token. Therefore, our algorithm ensures that

$$session(\alpha) = session(\beta) \implies type(\alpha) = type(\beta). \quad (4)$$

We use $cs_i$ to denote the fact that process $p_i$ is currently in the critical section. Note that a process cannot use its token to enter the critical section until the token has become safe. Therefore, we have,

$$(p_i \text{ is holding } \alpha) \wedge cs_i \implies safe(\alpha). \quad (5)$$

A token belonging to a session becomes safe only after all token belonging to the previous session have been released. Therefore,

$$session(\alpha) < session(\beta) \implies \neg safe(\beta). \quad (6)$$

We now prove that our algorithm is safe.

**Theorem 1** (*Group mutual exclusion*). *If two processes are concurrently executing their critical sections, then both critical sections are of the same type. Formally,*

$$cs_i \wedge cs_j \wedge (p_i \text{ is holding } \alpha) \wedge (p_j \text{ is holding } \beta) \implies type(\alpha) = type(\beta).$$

```
Group mutual exclusion algorithm for process p_i (continued):

(A1) On generating request for critical section of type x:
    if (token_i ≠ null) and ((token_i.session = 0) ∨ (token_i.type = x)) then
        if token_i.session = 0 then          // process p_i is the initial token holder
            token_i.session := 1;            // use the token without sending REQUEST messages
            token_i.type := x;
            token_i.safe := true;
        endif;
        token_i.idle := false;
        enter the critical section;
    else
        ++request_i[i].number;
        request_i[i].type := x;
        broadcast REQUEST(request_i[i].number, x) to all processes;
    endif;

(A2) On receiving REQUEST(number,x) message from process p_j:
    if request_i[j].number < number then // update request_i vector, if new request
        request_i[j].number := number;
        request_i[j].type := x;
    endif;
    y := request_i[j].type;
    if (token_i ≠ null) and (request_i[j].number > token_i.fulfilled[j]) then
        // process p_i is a token holder and process p_j's request is a new request
        if isPrimary_i then
            if (token_i.session > 0) and (token_i.type = y) then
                // send a secondary token to process p_j
                send TOKEN(token_i, false) to process p_j;
                ++token_i.fulfilled[j];
                ++token_i.noOfTokensIssued;
            else
                add (j, y) to token_i.queue, if not already present;
                if token_i.idle then call sendToken( ) endif;
            endif;
        else if (token_i.type ≠ y) and token_i.idle then
            broadcast RELEASE(token_i.session) to all processes;
            token_i := null;
        endif;
    endif;
```

Fig. 2. Token-based group mutual exclusion algorithm (continued).

**Proof.** We have,

$$cs_i \wedge cs_j \wedge (p_i \text{ is holding } \alpha) \wedge (p_j \text{ is holding } \beta)$$
$$\Longrightarrow \{ \text{using (5)} \}$$
$$safe(\alpha) \wedge safe(\beta)$$
$$\Longrightarrow \{ \text{using (6)} \}$$
$$session(\alpha) = session(\beta)$$
$$\Longrightarrow \{ \text{using (4)} \}$$
$$type(\alpha) = type(\beta)$$

This establishes the safety property. □

We now show that our algorithm is live. A process broadcasts its request for critical section to all processes in the system unless it already has an appropriate token. It can be easily shown that:

**Lemma 2.** *Every request for critical section that is broadcast to other processes is eventually added to the queue of the primary token.*

Moreover, a process holding a token releases it once it learns of a conflicting pending request and the token becomes idle. As a result,

**Lemma 3.** *Assume that there is session in progress. If there is conflicting pending request in the system, then the current session eventually terminates.*

It now suffices to show that if the queue of the primary token contains a pending request of type $t$, then a session of type $t$ is initiated eventually.

For a session with sequence number $s$ (hereafter, simply referred to as session $s$), let $queue(s)$ denote the set of types for which there is at least one request in the queue of the primary token when the type/group for session $s$ is selected. Also, for a type $t$, let $priority(t, s)$ denote the priority of type $t$ at the time of the selection. In case there is no request for critical section of type $t$ in the queue, we define $priority(t, s)$ to be zero. Clearly,

$$t \in queue(s) \equiv priority(t, s) > 0. \quad (7)$$

```
Group mutual exclusion algorithm for process p_i (continued):

(A3) On receiving TOKEN(token, isPrimary) message:
    token_i := token;
    token_i.idle := false;
    isPrimary_i := isPrimary;
    token_i.safe := token_i.safe ∨ safe_i;
    // does process p_i know of any pending request for the same type of critical section as its own?
    for k ∈ [1..n] do
        if isPrimary_i and
           (request_i[k].number > token_i.fulfilled[k]) and (token_i.type = request_i[k].type) then
            // send a secondary token to process p_k
            ++token_i.fulfilled[k];
            ++token_i.noOfTokensIssued;
            send TOKEN(token_i, false) to process p_k;
        endif;
    endfor;
    if token_i.safe then enter the critical section; endif;

(A4) On leaving the critical section:
    token_i.idle := true;
    if isPrimary_i then
        for each k ∈ [1..n] do              // add any new requests to the queue
            if request_i[k].number > token_i.fulfilled[k] then
                add (k, request_i[k].type) to token_i.queue, if not already present;
            endif;
        endfor;
        call sendToken( );                  // start a new session, if possible
    else
        // release the secondary token if there is a conflicting pending request
        if ∃k :: (request_i[k].number > token_i.fulfilled[k])∧
                 (request_i[k].type ≠ token_i.type) then
            broadcast RELEASE(token_i.session) to all processes;
            token_i := null;
        endif;
    endif;
```

Fig. 3. Token-based group mutual exclusion algorithm (continued).

Let $type(s)$ denote the type for session $s$. Note that, once there is a pending request of type $t$ in the queue of the primary token, the priority of $t$ increases monotonically until a session of type $t$ is initiated. Let $\delta_{\min}$ denote the minimum amount by which the priority of a type increases between two consecutive sessions. Further, let $\delta_{\max}$ denote the greater of (1) the maximum amount by which the priority of a type can increase between two consecutive sessions, and (2) the maximum value that the priority of a type can assume for the session immediately following a session in which that type was selected. Formally,

$$\begin{array}{c}(t \in queue(s)) \wedge (type(s) \neq t)\\ \Longrightarrow \\ 0 < \delta_{\min} \leqslant priority(t, s+1) - priority(t, s) \leqslant \delta_{\max}\end{array} \tag{8}$$

and,

$$\begin{array}{l}(t \in queue(s)) \ \wedge (type(s) = t)\\ \qquad \Longrightarrow \ priority(t, s+1) \leqslant \delta_{\max}.\end{array} \tag{9}$$

For our priority-based scheme, $\delta_{\min} = 1$ and $\delta_{\max} = n$. Evidently, (8) and (9) imply that, for all $x \geqslant 0$:

$$type(s) = t \implies priority(t, s+x) \leqslant x\delta_{\max}. \tag{10}$$

The following lemma can be easily proved:

**Lemma 4.** *Assume that there is a pending request of type $t$ in the queue of the primary token. Then, eventually either a session of type $t$ is initiated or the priority of $t$ becomes sufficiently large (specifically, at least $m\delta_{\max}$). Formally,*

$$t \in queue(s) \implies \langle \exists\, x : x \geqslant 0 : (type(s+x) = t) \vee (priority(t, s+x) \geqslant m\delta_{\max}) \rangle$$

**Proof.** Follows from the fact that, whenever a new session is initiated, the priority of $t$ increases by at least $\delta_{\min}$, where $\delta_{\min} > 0$, unless the type of the new session is $t$. □

**Lemma 5.** *Once the priority of a type becomes sufficiently large (specifically, at least $m\delta_{\max}$), at most $m-1$ sessions can be initiated before a session of type $t$ has to be initiated. Formally,*

$$priority(t, s) \ \geqslant \ m\delta_{\max} \implies \langle \exists\, x : 0 \leqslant x < m : type(s+x) = t \rangle.$$

```
Group mutual exclusion algorithm for process p_i (continued):

(A5) On receiving RELEASE(session) message from process p_j:
    if session_i < session then
        session_i := session;
        noOfReleaseMessages_i := 1;
    else if session_i = session then ++noOfReleaseMessages_i; endif;
    if (token_i ≠ null) and not(token_i.safe) then
        if safe_i then
            token_i.safe := true;
            token_i.idle := false;
            enter the critical section;
        endif;
    endif;

(A6) On invocation of sendToken( ):
    if token_i.queue is non-empty then
        // group the outstanding requests based on their type
        pendingTypes := { y | ∃k :: (k, y) ∈ token_i.queue };
        compute the priority for each type in pendingTypes;
        x := type with highest priority in pendingTypes;
        S := { p_k | (k, x) ∈ token_i.queue };
        let p_j be some process in S;
        // next session will be of type x with process p_j as the primary token holder
        // other processes in S will receive a secondary token
        for each p_k ∈ S do
            remove (k, x) from token_i.queue;
            ++token_i.fulfilled[k];
        endfor;
        token_i.oldNoOfTokens := token_i.noOfTokensIssued;
        ++token_i.session;
        token_i.noOfTokensIssued := |S| − 1;
        token_i.safe := safe_i;
        // send the primary token to process p_j
        send TOKEN(token_i, true) to process p_j;
        // send a secondary token to other processes in S
        send TOKEN(token_i, false) to each process p_k in S \ {p_j};
        token_i := null;
    endif;
```

Fig. 4. Token-based group mutual exclusion algorithm (continued).

| | | |
|---|---|---|
| $session(\alpha)$ | $\triangleq$ | sequence number carried by the token $\alpha$ |
| $type(\alpha)$ | $\triangleq$ | type of the token $\alpha$ |
| $safe(\alpha)$ | $\triangleq$ | the token $\alpha$ is safe to use |

Fig. 5. Notation used in the proof.

**Proof.** Assume that the next $m-1$ sessions that are initiated are of type different from $t$. We first show that all $m-1$ sessions are of different types. Assume, on the contrary, that two sessions $s+i$ and $s+j$ are of the same type, say $u$, where $0 \leqslant i < j < m-1$. Since $type(s+i) = u$, from (10),

$$priority(u, s+j) \leqslant (j-i)\,\delta_{\max} < m\delta_{\max}. \tag{11}$$

However, we have:

$$priority(t, s+j) > priority(t, s) \geqslant m\delta_{\max}. \tag{12}$$

In other words, $priority(u, s+j) < priority(t, s+j)$. Therefore, the type of session $s+j$ cannot be $u$ because another type has higher priority than $u$. We next show that the type $t$ has the maximum priority among all types when selecting type for session $s+m-1$. Clearly, from the above argument, all types except $t$ are selected exactly once in sessions $s, s+1, \ldots, s+m-2$. Therefore, using (8) and (10), for any type $u \neq t$, we have:

$$\begin{aligned} priority(u, s+m-1) &\leqslant m\delta_{\max} \leqslant priority(t, s) \\ &< priority(t, s+m-1). \end{aligned}$$

As a result, when selecting the type for session $s+m-1$, type $t$ has the maximum priority among all types. □

We now prove the liveness of our algorithm.

**Theorem 6** (*Starvation freedom*). *Every request for critical section is eventually fulfilled.*

**Proof.** Follows from Lemmas 2–5. Specifically, using Lemmas 4 and 5, it can be shown that at most $m(\frac{\delta_{\max}}{\delta_{\min}}+1)-1$ sessions can be initiated before a session of the same type as the request pending in the token queue is initiated. □

Finally, we show that our algorithm satisfies the concurrent entry property. Observe that if all requests for critical section are of the same type and no conflicting request is ever generated, then eventually all processes with outstanding requests eventually receive either a primary or a secondary token within one message hop of the termination of the previous session. Thus, we have,

**Theorem 7** (*Concurrent entry*). TokenGME *satisfies the concurrent entry property.*

### *3.5. Performance analysis*

In this section, we analyze the performance of our algorithm with respect to various complexity measures.

Our algorithm exchanges three kinds of messages, namely REQUEST, RELEASE and TOKEN. Clearly, for each request, there are at most $n-1$ REQUEST messages, at most $n-1$ RELEASE messages and at most 1 TOKEN message. We have,

**Theorem 8** (*Message complexity*). *The worst-case message complexity of* TokenGME *is* $2n-1$.

All messages except the TOKEN message are of size $O(1)$; the TOKEN message is of size $O(n)$. Therefore,

**Theorem 9** (*Message-size complexity*). *The amortized message-size complexity of* TokenGME *is* $O(1)$.

Assume that the system is heavily loaded. A process is already aware of a conflicting request when it leaves its critical section. Once the primary token holder of the current session chooses a new primary token holder, the next session starts as soon as the new primary token holder has (1) received the primary token from the primary token holder of the current session, and (2) received RELEASE messages from all secondary token holders of the current session. Thus,

**Theorem 10** (*Synchronization delay*). *The synchronization delay of* TokenGME *is one message hop.*

Now, assume that the system is lightly loaded. A process, on generating a request for critical section, can enter the critical section as soon as (1) its REQUEST message has been received by the current primary token holder, which in turn sends a token to it, and (2) it has received the token sent to it. Hence,

**Theorem 11** (*Waiting time*). *The waiting time of our algorithm is two message hops.*

Finally, if all processes request critical section of the same type and no conflicting request is generated for a sufficiently long time, then all processes eventually receive a primary or a secondary token. Therefore,

**Theorem 12** (*Maximum concurrency*). *The maximum concurrency of* TokenGME *is* $n$.

## 4. Experimental results

We experimentally compare the performance of TokenGME with Joung's second algorithm RA2 using an event-based simulation. We compare the performance of the two algorithms with respect to four metrics, namely message complexity, message-size complexity, waiting time and system throughput. To make it easier to compare the two algorithms, we report the ratio $\frac{\text{TokenGME's performance}}{\text{RA2's performance}}$ for each metric. Note that, for message complexity, message-size complexity and waiting time, a ratio of less than one would imply that TokenGME has better performance than RA2. On the other hand, for system throughput, a ratio of greater than one would imply that TokenGME has better performance than RA2. Later, we also compare the performance of the priority-based scheme for selecting the type of the next session with a simple FIFO-based scheme. In the FIFO-based scheme, the type for the next session is simply the type for the first outstanding request in the queue. Again, for ease of comparison, we report the ratio $\frac{\text{priority-based scheme's performance}}{\text{FIFO-based scheme's performance}}$ for each of the four metrics. Each point in our graph is averaged over multiple runs to obtain 95% confidence level.

In RA2, many messages need to carry a vector of size $n$. We use Singhal and Kshemkalyani's scheme [25] to reduce the size of messages in RA2. The main idea behind the scheme, which assumes FIFO channels, is as follows: a process, when sending a message carrying a vector to another process, piggybacks only those entries of the vector on the message that have changed since it last sent a message to that process.

Our experimental study has the following parameters:

- Number of processes, denoted by $n$.
- Number of groups, denoted by $m$.
- Mean duration of non-critical section or inter-request delay, denoted by $\mu_{ncs}$. We assume that inter-request delay is exponentially distributed with mean $\mu_{ncs}$.
- Mean duration of critical section, denoted by $\mu_{cs}$. We assume that critical section duration is uniformly distributed in the range $[0, 2*\mu_{cs}]$.
- Mean transmission or channel delay, denoted by $\mu_{cd}$. We assume that channel delay is exponentially distributed with mean $\mu_{cd}$.
- Degree of non-uniformity of group access distribution modeled using two parameters $\alpha$ and $\beta$. We assume that $\alpha$% of the groups are selected $\beta$% of the times.

In addition to the above-described six parameters, there are two additional parameters in our study, namely channel bandwidth and number of critical section requests per process. Both

of these parameters, however, have fixed values in our experiments. Specifically, we set channel bandwidth to 100 Kb per unit time. (For 1 time unit = 1 ms, this would correspond to channel bandwidth of 100 Mbps.) Further, each process makes 500 requests for critical section.

### 4.1. TokenGME *versus* RA2

We now vary the six parameters one-by-one to study their impact on the relative performance of TokenGME and RA2. Fig. 6(a)–(f) depicts the variation in the ratios for the four metrics as a function of various parameters.

Our experiments indicate that TokenGME almost always has better performance than RA2 except when the number of groups is two or the number of processes is very small (fewer than 10). In the first case, RA2 has around 10% lower waiting time and 5% higher system throughput than TokenGME. In the second case, RA2 has better message-size complexity than TokenGME. In all other cases, TokenGME outperforms RA2 in terms of message-size complexity, waiting time and system throughput, and, in one case, in terms of message complexity as well.

As expected, the message-size complexity of TokenGME is much lower than that of RA2 and the ratio primarily depends on the number of processes in the system. For instance, as depicted in Fig. 6(f), when the number of processes increases from 5 to 500, the ratio of the message-size complexities of the two algorithms decreases from 2.22 to 0.01. In all other cases, the ratio stays around 0.05 irrespective of the value of the parameter being varied. Note that TokenGME has significantly lower message-size complexity than RA2 even though we used Singhal and Kshemkalyani's scheme [25] to reduce the size of messages in RA2.

As our simulation results demonstrate, TokenGME can decrease the waiting time of a request by as much as 50% when compared to RA2. Further, the ratio of the waiting times of the two algorithms decreases when either (1) the mean inter-request delay increases (see Fig. 6(b)), or (2) the mean duration of critical section decreases (see Fig. 6(c)), or (3) the mean channel delay increases (see Fig. 6(d)), or (4) the degree of non-uniformity decreases (see Fig. 6(e)), or (5) the number of processes increases (see Fig. 6(f)). An analogous trend is observed with respect to system throughput—albeit in opposite direction—except when mean inter-request delay is varied. In contrast to the waiting time ratio, which decreases significantly with the increase in mean inter-request delay, the system throughput ratio actually tends to 1.

Finally, TokenGME has only slightly better message complexity than RA2 in almost all cases except when mean inter-request delay is varied. As the mean inter-request delay increases, the message complexity ratio decreases by as much as 45%.

### 4.2. Priority-based scheme versus FIFO-based scheme

To compare the performance of the priority-based scheme with the simple FIFO-based scheme when selecting the type for the next session, we vary the number of processes and the degree of non-uniformity one-by-one. The results are shown in Fig. 7(a) and (b). Our experiments indicate that the priority-based scheme exhibits lower waiting time and higher system throughput than the FIFO-based scheme as the number of processes increases or the group access distribution becomes more non-uniform. Specifically, as the number of processes increases, the priority-based scheme decreases the waiting time by as much as 24% and increases the system throughput by as much as 22% when compared to the FIFO-based scheme. Further, as the degree of non-uniformity increases, the priority-based scheme decreases the waiting time by as much as 18% and increases the system throughput by as much as 14% when compared to the FIFO-based scheme.

### 4.3. Discussion

There are many reasons for the better performance of TokenGME as compared to RA2.

First, RA2 uses logical timestamp to select the type for the next session, whereas TokenGME uses a priority-based scheme to select the type for the next session. (Note that, in RA2, the type selection for the next session occurs implicitly via logical timestamps rather than explicitly as in TokenGME.) The priority-based scheme assigns higher weightage to types with larger number of outstanding requests. This helps to improve the performance of TokenGME as illustrated by the following example. Suppose there are six pending requests for critical section generated at around the same time. Further, suppose two requests are of one type, say $x$ and the remaining four are of another type, say $y$. With RA2, the next session may be of either type depending on which request has the smallest timestamp. In case type $x$ is picked over type $y$, four requests are forced to wait for two requests to be fulfilled. However, with TokenGME, type $y$ is picked over type $x$. This ensures that only two requests are forced to wait for four requests to be fulfilled. Clearly, this results in lower waiting time as compared to the alternative.

Second, in RA2, when a process generates a request for critical section, it sends a REQUEST message to all other processes in the system, and waits to receive a REPLY message from them. Once the process has received a REPLY message from all other processes in the system, it enters its critical section as a *leader*. As a leader, it may invite other processes that have outstanding requests compatible with its own request to enter their critical sections as *followers*. Specifically, to enter the critical section as a leader, a process has to wait to receive a REPLY message from *all other processes in the system*. Long transmission delay in any one of the REPLY messages can delay the entry of the process into its critical section. In TokenGME, on the other hand, a process with an unsafe token has to wait to receive a RELEASE message only from *all token holders of the previous session* to enter its critical section. Long transmission delay in any one of the RELEASE messages can delay entry of the process into its critical section. Typically, the set of token holders of a session is much smaller than the entire set

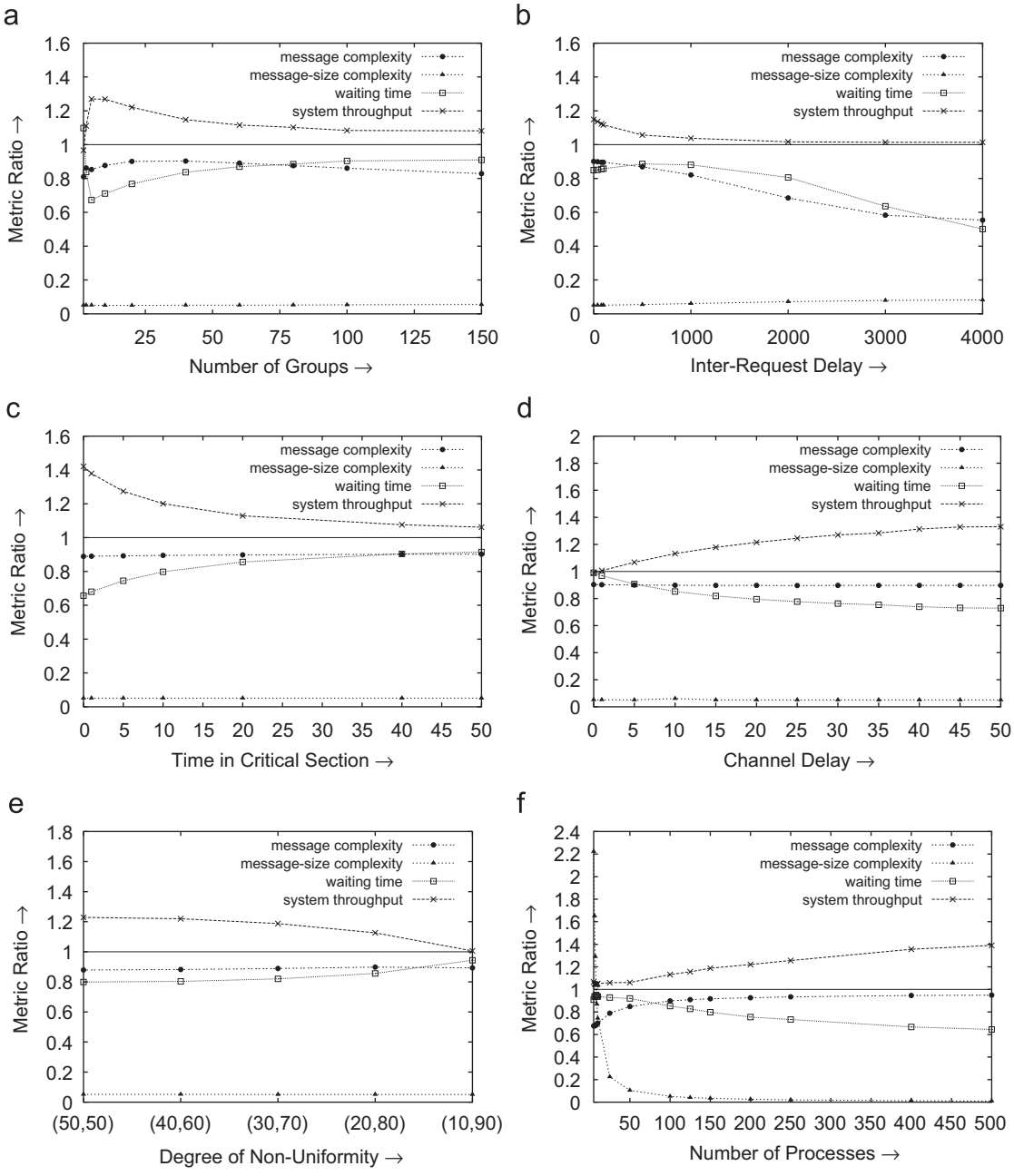


Fig. 6. Relative performance of TokenGME with respect to RA2 as a function of various parameters: (a) varying the number of groups $m$ from 2 to 150 with $n = 100$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units, $\mu_{cd} = 10$ time units and $(\alpha, \beta) = (20, 80)$. (b) Varying the mean inter-request delay $\mu_{ncs}$ from 0 to 4000 time units with $n = 100$, $m = 50$, $\mu_{cs} = 20$ time units, $\mu_{cd} = 10$ time units and $(\alpha, \beta) = (20, 80)$. (c) Varying the mean time in critical section $\mu_{cs}$ from 0 to 50 time units with $n = 100$, $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cd} = 10$ time units and $(\alpha, \beta) = (20, 80)$. (d) Varying the mean channel delay $\mu_{cd}$ from 0 to 50 time units with $n = 100$, $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units and $(\alpha, \beta) = (20, 80)$. (e) Varying the degree of non-uniformity $(\alpha, \beta)$ from $(50, 50)$ to $(10, 90)$ with $n = 100$, $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units and $\mu_{cd} = 10$ time units. (f) Varying the number of processes from 5 to 500 with $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units, $\mu_{cd} = 10$ time units, and $(\alpha, \beta) = (20, 80)$.

of processes. As a result, the probability that one of the REPLY messages experiences a long transmission delay is much higher than the probability that one of the RELEASE messages experiences a large transmission delay.

Third, in RA2, once all processes in the current session have left their critical sections, there is delay of *at least two message hops* before a follower in the next session can enter its critical section. On the other hand, in TokenGME, once all processes in the current session have left their critical section, a secondary token holder of the next session may be able to enter its critical section only after a delay of one message hop. This happens, for instance, if the secondary token holder of a session receives the token directly from the primary token holder of the previous session.

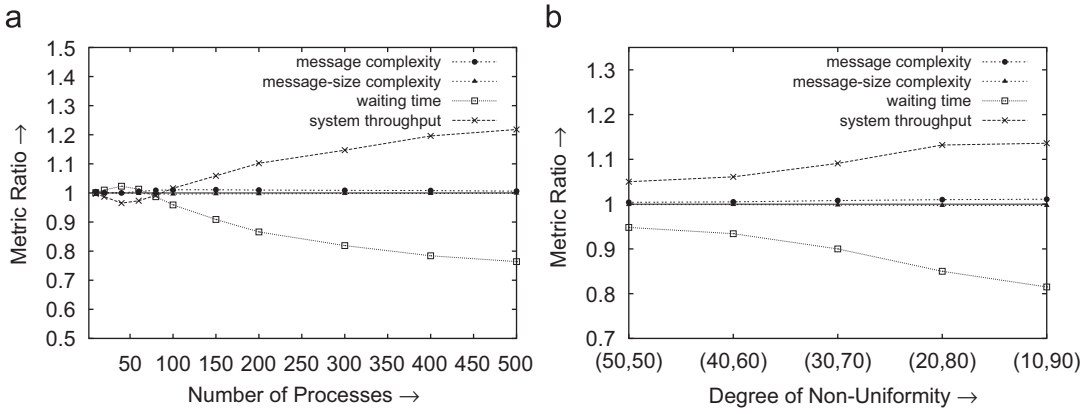


Fig. 7. Relative performance of the priority-based scheme with respect to the FIFO-based scheme: (a) varying the number of processes $n$ from 2 to 500 with $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units, $\mu_{cd} = 10$ time units and $(\alpha, \beta) = (20, 80)$. (b) Varying the degree of non-uniformity $(\alpha, \beta)$ from $(50, 50)$ to $(10, 90)$ with $n = 200$, $m = 50$, $\mu_{ncs} = 50$ time units, $\mu_{cs} = 20$ time units and $\mu_{cd} = 10$ time units.

## 5. Enhancements to the basic algorithm

In this section, we discuss several extensions to TokenGME to further improve its performance and/or make it fault-tolerant.

### 5.1. Achieving unnecessary blocking freedom

Consider the CD jukebox example. Suppose some data are replicated on multiple CDs. Therefore a request for such data can be satisfied using any *one* of the CDs on which data has been replicated. In traditional group mutual exclusion, a process has to specify the type of critical section it wants to execute at the time of the request—which translates into the CD it wants to access for satisfying its request. This may lead to *unnecessary delay* (*or blocking*) in satisfying a request. To eliminate this unnecessary delay, Manabe and Park extend the group mutual exclusion problem in [18] to allow a process to specify more than one type when making a request. The request can be satisfied by allowing a process to execute critical section for any one of those types (specified at the time of the request). TokenGME can be easily modified to achieve unnecessary blocking freedom. For a request $r$, let $types(r)$ denote the set of types specified with the request.

Recall that, when selecting the type for a session, the set of all outstanding requests is divided into subsets based on request type. Since, at the time of making a request, a process can specify more than one type, an outstanding request can now belong to multiple subsets. Specifically, a pending request $r$ belongs to all subsets corresponding to the types in $types(r)$. A subset is assigned a priority as described earlier in Section 3.2.1 and the rest of the steps remain the same.

We also extend the notions of compatible and conflicting requests as follows. A request $r$ is said to be compatible with a type $t$ if $t \in types(r)$; otherwise $r$ is said to conflict with $t$. Now, a primary token holder, on receiving a request compatible with the type of the token, issues a secondary token to the requesting process unless there is a conflicting request already present in the primary token queue.

It can be verified that the modified algorithm satisfies the following property: if all pending requests in the system are compatible with the type of the current session in progress and no conflicting request is ever generated, then all requesting processes enter their critical sections within two message hops (of making the request). (This property is analogous to the concurrent entry property of traditional group mutual exclusion problem.) Moreover, all complexity measures of the algorithm remain the same except for message-size complexity. The (amortized) message-size complexity increases to $O(g)$, where $g$ is the maximum number of types or groups that a process can specify at the time of making a request.

### 5.2. Bounded implementation of TokenGME

TokenGME uses two kinds of integers whose value may grow without any bound: the sequence number for a request and the sequence number for a session. There are $n$ integers of the former kind, one stored at each process, and one integer of the latter kind, stored in the primary token. To obtain an implementation in which all integers are bounded, we can use the approach described in [27]. Let $s_{\max}$ denote the maximum value that a sequence number can assume, where $s_{\max}$ is at least 2. The main idea is to reset a sequence number to 0 once its value reaches $s_{\max}$.

#### 5.2.1. Resetting sequence number for request

The sequence number at a process can be reset as follows [27]. Consider a process $p_i$ whose sequence number has reached $s_{\max}$. Once $p_i$'s request with sequence number $s_{\max}$ has been fulfilled, it broadcasts a RESET_RSN message to all processes instructing them to reset the $i$th entry in their request vectors to 0. Further, it stops generating any further requests until the reset operation is complete. A process $p_j$, on receiving a RESET_RSN message from $p_i$, waits until it has received all REQUEST messages that $p_i$ has sent so far since the beginning or the last reset operation (whatever the case may be). (The number of such REQUEST messages

should be exactly $s_{\max}$.) Once that happens, there should not be any REQUEST messages in transit from $p_i$ to $p_j$. Process $p_j$ then resets the $i$th entry in its request vector and sends an acknowledgment message to $p_i$ informing it of the same. Finally, once $p_i$ has received an acknowledgment message from all processes, it can resume generating requests for critical section. Note that it is not necessary to reset the $i$th entry in the fulfilled vector of a token. For the purpose of determining whether a request is new (which is done by comparing the appropriate corresponding entries of request and fulfilled vectors), we assume that $s_{\max} < 1$. Further, when an entry of the fulfilled vector is incremented after reaching the maximum value, the new value of the entry is set to 1.

*5.2.2. Resetting sequence number for session*

A similar approach can be used to reset the sequence number for a session. Recall that session number is used to determine the session to which a RELEASE message belongs. This, in turn, helps a process to determine whether all secondary tokens issued in the previous session have been released. Suppose the sequence number for session has reached $s_{\max}$. Before starting the next session, the current primary token holder, say process $p_i$, broadcasts a RESET_SSN message to all processes instructing them to reset their session numbers to 0. The reset message contains an integer *totalNoOfTokensIssued* to indicate the *total* number of secondary tokens that have been issued so far since the beginning or the last reset operation (whatever the case may be). A process $p_j$, on receiving a RESET_SSN message, waits until it has received the matching number of RELEASE messages. At this time, $p_j$ should have received RELEASE messages for all secondary tokens issued so far. It then resets its session number and sends an acknowledgment message to $p_i$ informing it of the same. Finally, once $p_i$ has received an acknowledgment message from every process, it initiates the next session. Note that the value of the integer *totalNoOfTokensIssued* is bounded by $ns_{\max}$ assuming that at most one secondary token is issued to each process in a session. TokenGME can be modified easily to ensure this property without affecting any complexity measure.

*5.2.3. Bounds on various integers*

It can be verified that various integers used in the modified algorithm have maximum values as shown below:

- *oldNoOfTokens*, *noOfTokensIssued*, *noOfReleaseMessages*: $n$;
- entry of $request$ vector, entry of *fulfilled* vector, *session*: $s_{\max}$;
- *totalNoOfTokensIssued*: $ns_{\max}$;
- $age$, *priority*: $m(\frac{\delta_{\max}}{\delta_{\min}} + 1)$.

*5.3. Achieving fault tolerance*

In the paper, so far we have assumed that both processes and channels are reliable. In case one or more processes may fail by crashing, TokenGME may not work correctly. Specifically, one or more tokens may get lost as a result of which the system becomes deadlocked. We now describe an approach that can be used to tolerate process crashes. The main idea is to *restart* TokenGME on the set of processes that are still operational. A simple mechanism is used to fulfill requests that are still pending when TokenGME is restarted. We do not, however, restart the underlying application and assume that it is capable of coping with process crashes. Note that the underlying application generates critical section requests and also executes critical sections. A group mutual exclusion algorithm is only responsible for informing a process when it can enter the critical section and also reacting to when a process leaves the critical section. In case application has to be restarted as well, then Arora and Gouda's approach can be used to reset the entire system [1].

*5.3.1. Failure detection*

We assume that processes are equipped with a *perfect failure detector* [5] that can be used to *reliably* detect failure of a process. It can be shown that a crash-tolerant mutual exclusion (and, therefore, group mutual exclusion) algorithm can be used to implement a perfect failure detector provided it is possible to "peek inside" the algorithm [8]. This, in turn, implies that a perfect failure detector is *necessary* to solve the group mutual exclusion problem in an environment where processes may fail by crashing.

*5.3.2. Failure recovery*

To recover from process failures, one of the operational processes in the system is chosen to act as a *coordinator*, which is then responsible for starting a new instance of the group mutual exclusion algorithm on the set of processes that are still operational. A simple way to choose a coordinator is to select the process with the smallest identifier among all operational processes. Specifically, whenever a process detects crash of another process, it sends a NOTIFY message to the process that has the smallest identifier among all operational processes in its view. The NOTIFY message is piggybacked with the set of all processes that have failed so far in its view. Further, the process that has the smallest identifier among all processes currently operational in its view (including itself) elects itself as the coordinator. The coordinator waits until it has received a NOTIFY message from all operational processes indicating that all of them agree on the set of failed processes. It then restarts TokenGME in three phases.

In the *first* phase, the coordinator instructs all processes that are still alive to abort the current instance of TokenGME and start a new instance of TokenGME via a RESTART message. The new instance involves only those processes that are still alive. (In case a process is already in the critical section when it receives a RESTART message, it defers the processing of the RESTART message until it has left the critical section.) Note that two operational processes may detect crash of different processes in different order. To ensure that all operational processes agree on the set of processes participating in the current instance of TokenGME, the coordinator piggybacks the set of processes that have failed in its view on the RESTART message. Further, the execution of the new instance of TokenGME

is *suspended* until a RESUME message is received from the coordinator, which happens in the third phase. Once a process has started a new instance of TokenGME, it sends a DONE message to the coordinator. In case the process has a pending request at the time of restarting, it piggybacks the pending request on the DONE message. We refer to such a request as *old request*. The first phase ends when the coordinator has received a DONE message from all operational processes.

In the *second* phase, the coordinator collects all old requests it received along with DONE messages and constructs a deterministic schedule to fulfill the old requests (the details are explained later). The second phase ends once the coordinator learns that all old requests have been fulfilled.

Finally, in the *third* phase, the coordinator broadcasts a RESUME message to all operational processes instructing them to *resume* the execution of the instance of TokenGME started in the first phase.

When TokenGME is restarted, messages sent by processes before starting the new instance of TokenGME may still be in transit. Such messages should be ignored on arrival. To enable a process to identify such messages, each message sent by a process is piggybacked with an *instance identifier*, which, for example, can be the number of processes participating in that instance. Note that if a process fails before the three phases of a restart operation have finished, then a new coordinator is elected, if required, which then initiates a new restart operation. Therefore the restart operation itself can tolerate process failures.

*Fulfilling old requests for critical section*: Let $Q$ denote the set of processes that have old requests for critical section. Also, let $Q_1, Q_2, \ldots, Q_r$ be the partition of $Q$ based on the type of request a process has made. In other words, requests by all processes in $Q_i$, where $1 \leqslant i \leqslant r$, are of the same type. A simple way to fulfill old requests is to fulfill requests by processes in $Q_1$, followed by processes in $Q_2$ and so on. To that end, the coordinator sends a START message to all processes in $Q_1$. Once a process in $Q_i$, where $1 \leqslant i < r$, leaves its critical sections, it sends a LEAVE message to all processes in $Q_{i+1}$. A process in $Q_{i+1}$ enters its critical section after it has received a LEAVE message from all processes in $Q_i$. Finally, a process in $Q_r$ sends a LEAVE message to the coordinator after leaving its critical section. Clearly, once the coordinator has received a LEAVE message from all processes in $Q_r$, it knows that all old requests have been fulfilled.

### 5.3.3. Proving correctness

A formal description of the restart operation is given in Figs. 8–11. It consists of nine actions, namely B1–B6 and C1–C3. Actions B1–B6 are executed by all processes, whereas actions C1–C3 are executed only by the coordinator.

We now argue that our approach for achieving fault-tolerance is correct. To that end, we associate an attribute with every operational process referred to as its *tag*. The tag of a process $p_i$, denoted by $tag_i$, is a tuple consisting of two entries. The first entry is the identifier of the latest instance of TokenGME initiated on $p_i$. The second entry indicates whether the latest instance is running state or in suspended state. Specifically, the second entry has the value false if the latest instance (of TokenGME) is in suspended state; it has the value true otherwise. Two tags are compared lexicographically. Formally, given two tags $\langle x, a\rangle$ and $\langle y, b\rangle$:

$$\langle x, a\rangle < \langle y, b\rangle \quad \triangleq \quad (x \subsetneq y) \vee ((x = y) \wedge (a < b)).$$

Recall that instance identifier corresponds to the set of processes that have crashed so far. Further, we use the convention that false $<$ true. The following properties about the tag of a process can be easily verified:

$$\text{tag of a process is monotonically non-decreasing, and} \quad (13)$$

$$\text{tag of a process does not change while the process is in its critical section.} \quad (14)$$

To show that our fault-tolerant group mutual exclusion is crash-tolerant, we first prove the following lemma:

**Lemma 13.** *If two processes are executing their critical sections simultaneously, then they have identical tags. Formally,*

$$cs_i \wedge cs_j \quad \Longrightarrow \quad tag_i = tag_j$$

**Proof.** Assume, on the contrary, that $tag_i \neq tag_j$. Without loss of generality, assume that $tag_i < tag_j$. There are two cases to consider:

- *Case* 1 ($tag_i$ and $tag_j$ have the same instance identifier): Let the instance identifier for the two tags be $x$. Clearly, $tag_i = \langle x, \text{false}\rangle$ and $tag_j = \langle x, \text{true}\rangle$. This means that process $p_i$'s request is satisfied as old request during restart operation for instance $x$. This, in turn, implies that instance $x$ of TokenGME is in suspended state on all live processes including process $p_j$—a contradiction.
- *Case* 2: ($tag_i$ and $tag_j$ have different instance identifiers): Let $tag_i = \langle x, a\rangle$ and $tag_j = \langle y, b\rangle$. Observe that $x < y$ because $x \neq y$ and $tag_i < tag_j$. Since process $p_j$ is in critical section, all live processes including process $p_i$ have already sent a DONE message to the coordinator. This, in turn, implies that $p_i$ has already started instance $y$ of TokenGME—a contradiction.

This establishes the lemma. □

Clearly, if two critical section requests are satisfied by the same instance of TokenGME, then they satisfy the group mutual exclusion property. Likewise, if they are satisfied as old requests during the same restart operation, then they satisfy the group mutual exclusion property as well. Let $type_i$ denote the type of the critical section being executed by process $p_i$ if $p_i$ is in its critical section (and is undefined otherwise). It follows from Lemma 13 and our discussion that:

**Theorem 14** (*Group mutual exclusion*). *If two processes are executing their critical sections simultaneously, then the critical sections have the same type. Formally,*

$$cs_i \wedge cs_j \quad \Longrightarrow \quad type_i = type_j.$$

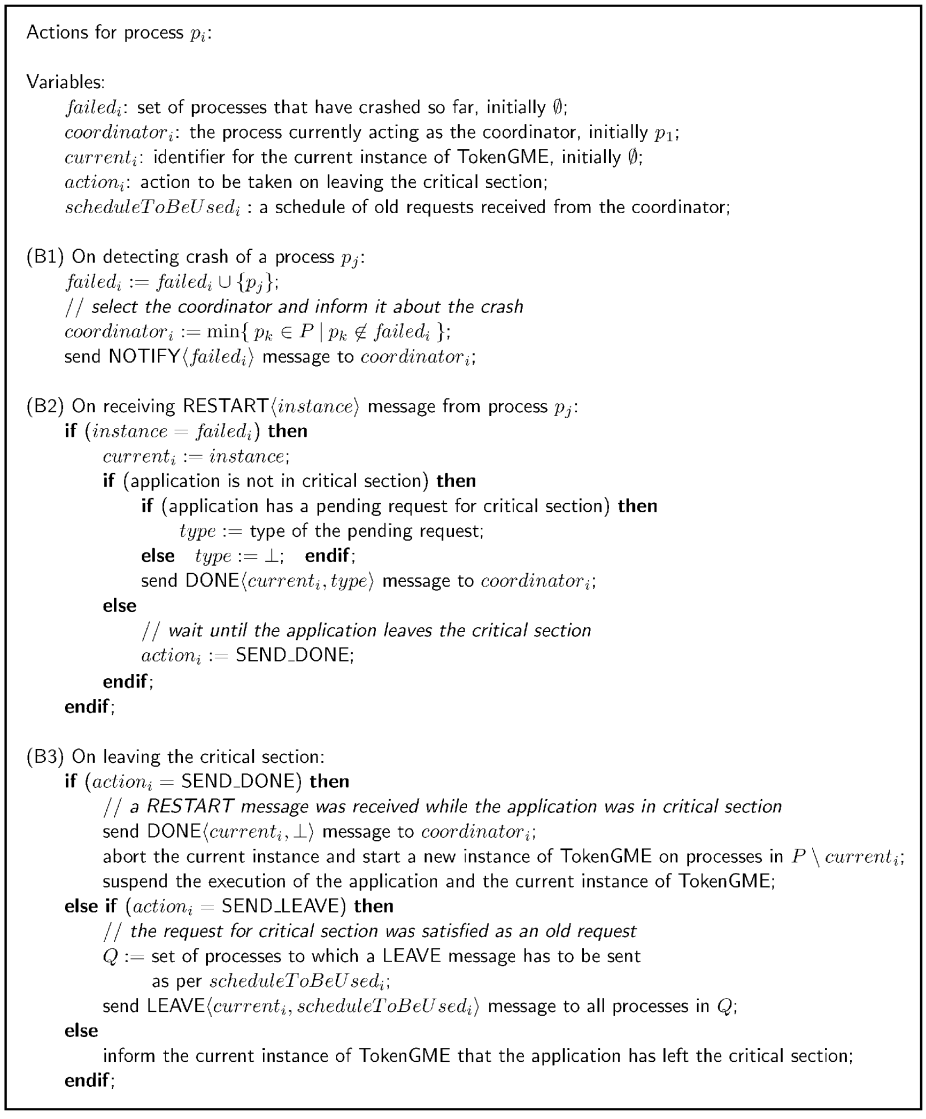


Fig. 8. Making the group mutual exclusion algorithm TokenGME fault-tolerant.

It now remains to be shown that our approach for achieving fault-tolerance satisfies the liveness property. A process is said to be *correct* if it never crashes. To show liveness, we have to show that every request for a critical section by a correct process is eventually fulfilled. Note that the system eventually reaches a state after which there are no more process crashes. Once that happens, an instance of TokenGME is eventually initiated on the set of correct processes, and, moreover, no new instance of TokenGME is initiated after that. If a correct process has an outstanding request for critical section at the time this final instance is initiated, the request is fulfilled as an old request. Otherwise, any request generated later is eventually satisfied via the final instance of TokenGME because TokenGME is live. Therefore, we have:

**Theorem 15** (*Starvation freedom*). *Every request by a correct process is eventually fulfilled.*

*5.3.4. Cost of failure recovery*

Each restart operation exchanges at most $5(n-1)$ messages: $n-1$ NOTIFY, $n-1$ RESTART, $n-1$ DONE, $n-1$ START and $n-1$ RESUME messages. For each request satisfied as a "normal" request, at most $2n-1$ messages are exchanged—$n-1$ REQUEST, 1 TOKEN and $n-1$ RELEASE messages. Further, for each request satisfied as "old" request, again at most $2n-1$ messages are exchanged—$n-1$ REQUEST, 1 TOKEN and $n-1$ LEAVE messages. Let $r_{total}$ denote the total number of requests generated during an execution. Also, let $f$ denote the total number of processes that fail during an execution. The total number of messages generated in the system is bounded by $(2n-1)r_{total} + 5(n-1)f$. In case $r_{total} \gg f$, overhead incurred due to restart operations can be ignored.

It is possible to eliminate the third phase in a restart operation altogether. When a new instance of TokenGME is started, its execution is not suspended but rather the instance is allowed

Actions for process $p_i$ (continued):

(B4) On receiving RESUME$\langle instance \rangle$ message from process $p_j$:
  **if** $(instance = current_i)$ **then**
    *// requests for critical section can now be managed by the group mutual exclusion algorithm*
    resume the execution of the application and the current instance of TokenGME;
  **endif**;

(B5) On receiving START$\langle instance, schedule \rangle$ message from process $p_j$:
  **if** $(instance = current_i)$ **then**
    $scheduleToBeUsed_i := schedule$;
    inform the application that it can enter the critical section;
    *// the process has to send a LEAVE message after leaving the critical section*
    $action_i :=$ SEND_LEAVE;
  **endif**;

(B6) On receiving LEAVE$\langle instance, schedule \rangle$ message from process $p_j$:
  **if** $(instance = current_i)$ **then**
    $scheduleToBeUsed_i := schedule$;
    $Q :=$ set of all processes from which a LEAVE message should be received
      as per $scheduleToBeUsed_i$;
    **if** (LEAVE message has been received from processes in $Q$) **then**
      inform the application that it can enter the critical section;
      *// the process has to send a LEAVE message after leaving the critical section*
      $action_i :=$ SEND_LEAVE;
    **endif**;
  **endif**;

Fig. 9. Making the group mutual exclusion algorithm TokenGME fault-tolerant (continued).

Actions for process $p_i$ on becoming the coordinator:

Variables:
  $othersFailed_i$: vector $[1..n]$ of set of failed processes according to each process, initially $[\emptyset, \ldots, \emptyset]$;
  $allFailed_i$: set of processes suspected by at least one process, initially $\emptyset$;
  $latest_i$: identifier for the latest instance of TokenGME, initially $\emptyset$;
  $oldRequests_i$: set of old requests in the system, initially $\emptyset$;
  $schedule_i$: a deterministic schedule of all old requests, initially $\emptyset$;

(C1) On receiving NOTIFY($failed\_rcvd$) message from process $p_j$:
  *// is it a new notification message?*
  **if** $(othersFailed_i[j] \subset failed\_rcvd)$ **then**
    $othersFailed_i[j] := failed\_rcvd$;
    $allFailed_i := allFailed_i \cup failed\_rcvd$;
    *// do all operational processes agree on the set of failed processes?*
    **if** $\langle \forall k : p_k \in P \setminus allFailed_i : othersFailed_i[k] = allFailed_i \rangle$ **then**
      $latest_i := allFailed_i$;
      $oldRequests_i := \emptyset$;
      *// instruct all operational processes to start a new instance of the*
      *// group mutual exclusion algorithm*
      send RESTART($latest_i$) to each process in $P \setminus latest_i$;
    **endif**;
  **endif**;

(C2) On receiving DONE$(instance, type)$ message from process $p_j$:
  **if** $(instance = latest_i)$ **then**
    **if** $(type \neq \perp)$ **then** add $\langle j, type \rangle$ to $oldRequests_i$; **endif**;
    **if** (DONE message has been received from all processes in $P \setminus latest_i$) **then**
      $schedule_i :=$ a deterministic schedule of all requests in $oldRequests_i$;
      **if** $(oldRequests_i \neq \emptyset)$ **then**
        *// there are one or more old requests for critical section in the system*
        send START$\langle latest_i, schedule_i \rangle$ message to all relevant processes as per $schedule_i$;
      **else**
        *// there are no old requests for critical section in the system*
        send RESUME$\langle latest_i \rangle$ message to all processes in $P \setminus latest_i$;
      **endif**;
    **endif**;
  **endif**;

Fig. 10. Making the group mutual exclusion algorithm TokenGME fault-tolerant (continued).

```
Actions for process p_i on becoming the coordinator (continued):

(C3) On receiving LEAVE⟨instance, _⟩ message from process p_j:
    if (instance = latest_i) then
        Q := set of all processes from which a LEAVE message is expected to be received
             as per schedule_i;
        if (LEAVE message has been received from all processes in Q) then
            // all old requests for critical section have been fulfilled
            send RESUME⟨latest_i⟩ message to all processes in P \ latest_i;
        endif
    endif;
```

Fig. 11. Making the group mutual exclusion algorithm TokenGME fault-tolerant (continued).

to execute normally. The coordinator holds the primary token initially. However, it does not issue any token to any process until all old requests have been fulfilled.

The second phase of the restart operation, strictly speaking, is not necessary. We use it to restrict the number of messages that are exchanged due to a request to $2n - 1$. Otherwise, a request may generate as many as $nf$ messages, in the worst case, which may be as large as $\Omega(n^2)$.

*5.3.5. Discussion*

The mechanism for achieving fault tolerance described in this section has two advantages. First, it is *fault reactive* in nature. This means that if no process fails during an execution then the performance of the fault-tolerant TokenGME is identical to that of fault-sensitive TokenGME. Further, the overhead incurred due to process failures increases in proportion to the number of processes that *actually fail* during an execution. Second, it is *general* in the sense that it can be used to transform *any* fault-sensitive (group) mutual exclusion algorithm into a fault-tolerant algorithm (and not just TokenGME).

For the traditional mutual exclusion problem, a common approach for achieving fault-tolerance when using a token-based mutual exclusion algorithm is to regenerate the token whenever its loss is detected (*e.g.*, [22,19,21,26]). Note that, in our algorithm, it is not sufficient to regenerate the primary token on detecting its loss. The regenerated primary token cannot be used until all secondary tokens in the system have been released. Otherwise, the safety property may be violated in case the newly generated primary token is used by a process whose request type is different from that of a secondary token.

Even though the failure-recovery mechanism presented here is general in the sense that it can be used with any (group) mutual exclusion algorithm, its performance (in terms of number of messages) is comparable with that of more specialized approaches (*e.g.*, [22,19,21]). For instance, Manivannan and Singhal's approach for regenerating a lost token may exchange $\Omega(n^2)$ messages whenever a process crashes [19]. (This happens when multiple processes with pending requests concurrently suspect that the token has been lost.) However, unlike our approach, their approach can recover from token lost due to channel failure. In Mueller's approach, a ring is used to detect whether a process has failed [21]. The approach works only if at most one process fails in the ring between two recovery operations. If two or more processes fail concurrently, then the ring may get partitioned into multiple segments. As a result, the fault handler may not receive local states of all operational processes and the system may not recover to a correct state.

## 6. Conclusion and future work

In this paper, we have proposed an efficient token-based distributed algorithm for solving the group mutual exclusion problem. Our algorithm is especially suited for applications in which some groups are requested more often than other groups. Our experimental results indicate that our algorithm has much better performance than existing group mutual exclusion algorithms especially when average channel delay is comparable to or larger than average duration of critical section. Specifically, for the parameter values for which we ran the experiments, it decreased the waiting time by as much as 35% and increased the system throughput by as much as 39% at the *same time* when compared to the algorithm proposed by Joung.

We have also described several modifications to the basic token-based algorithm to satisfy desirable properties such as unnecessary blocking freedom and bounded implementation. Further, we have presented a general mechanism to make the algorithm tolerant to process crashes.

Clearly, the scheme for selecting the type for the next session to be initiated is crucial to the performance of our algorithm. As future work, we plan to investigate other type-selection schemes and measure their performance experimentally.

## References

[1] A. Arora, M.G. Gouda, Distributed reset, IEEE Trans. Comput. 43 (9) (1994) 1026–1038.

[2] R. Atreya, N. Mittal, S. Peri, A Quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set, IEEE Trans. Parallel Distributed Syst. (TPDS), accepted for publication.

[3] J. Beauquier, S. Cantarell, A.K. Datta, F. Petit, Group mutual exclusion in tree networks, J. Inform. Sci. Eng. (JISE) 19 (3) (2003) 415–432.

[4] S. Cantarell, A.K. Datta, F. Petit, V. Villain, Token based group mutual exclusion for asynchronous rings, in: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), 2001, pp. 691–694.

[5] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.

[6] K.M. Chandy, J. Misra, The drinking philosophers problem, ACM Trans. Programming Languages Syst. (TOPLAS) 6 (4) (1984) 632–646.
[7] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1988.
[8] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, P. Kouznetsov, Mutual exclusion in asynchronous systems with failure detectors, J. Parallel Distributed Comput. (JPDC) 65 (4) (2005) 492–505.
[9] E.W. Dijkstra, Solution of a problem in concurrent programming control, Comm. ACM (CACM) 8 (9) (1965) 569.
[10] E.W. Dijkstra, Hierarchical ordering of sequential processes, Acta Inform. 1 (2) (1971) 115–138.
[11] M.J. Fischer, N.A. Lynch, J.E. Burns, A. Borodin, Resource allocation with immunity to limited process failure (preliminary report), in: Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS) October 1979, pp. 234–254.
[12] Y.-J. Joung, Asynchronous group mutual exclusion, Distributed Comput. (DC) 13 (4) (2000) 189–206.
[13] Y.-J. Joung, The congenial talking philosophers problem in computer networks, Distributed Comput. (DC) 2002, pp. 155–175.
[14] Y.-J. Joung, Quorum-based algorithms for group mutual exclusion, IEEE Trans. Parallel Distributed Syst. (TPDS) 14 (5) (2003) 463–475.
[15] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. ACM (CACM) 18 (8) (1974) 453–455.
[16] V. Madenur, N. Mittal, A delay-optimal group mutual exclusion algorithm for a tree network, J. Inform. Sci. Eng. (JISE), accepted for publication.
[17] M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, ACM Trans. Comput. Syst. 3 (2) (1985) 145–159.
[18] Y. Manabe, J. Park, A quorum-based extended group mutual exclusion algorithm without unnecessary blocking, in: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), Newport Beach, California, USA, 2004, pp. 341–348.
[19] D. Manivannan, M. Singhal, A decentralized token generation scheme for token-based mutual exclusion algorithms, Internat. J. Comput. Syst. Sci. Eng. 11 (1) (1996) 45–54.
[20] N. Mittal, P.K. Mohan, An efficient distributed group mutual exclusion algorithm for non-uniform group access, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, Phoenix, Arizona, USA, 2005, pp. 367–372.
[21] F. Mueller, Fault-tolerance for token-based synchronization protocols, in: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), 2001, pp. 1257–1264.
[22] M. Naimi, M. Trehel, How to detect a failure and regenerate the token in the $\log n$ distributed algorithm for mutual exclusion, in: Proceedings of the 2nd Workshop on Distributed Algorithms (WDAG), 1987, pp. 155–156.
[23] K. Raymond, A tree based algorithm for distributed mutual exclusion, ACM Trans. Comput. Syst. 7 (1) (1989) 61–77.
[24] G. Ricart, A.K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, Comm. ACM (CACM) 24 (1) (1981) 9–17.
[25] M. Singhal, A. Kshemkalyani, An efficient implementation of vector clocks, Inform. Process. Lett. (IPL) 43 (1992) 47–52.
[26] J. Sopena, L. Arantes, M. Bertier, P. Sens, A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree, in: Proceedings of the 11th European Conference on Parallel Computing (Euro-Par), 2005, pp. 654–663.
[27] I. Suzuki, T. Kasami, A distributed mutual exclusion algorithm, ACM Trans. Comput. Syst. 3 (4) (1985) 344–349.
[28] M. Toyomura, S. Kamei, H. Kakugawa, A Quorum-based distributed algorithm for group mutual exclusion, in: Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Chengdu, Sichuan, China, 2003, pp. 742–746.
[29] K.-P. Wu, Y.-J. Joung, Asynchronous group mutual exclusion in ring networks, IEEE Proceedings—Comput. Digital Tech. 147 (1) (2000) 1–8.

**Neeraj Mittal** received his B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a co-director of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking and databases.

**Prajwal K. Mohan** received his B.E. degree in computer science from B.M.S College of Engineering, Bangalore, India with distinction in 2003 and M.S. degree in computer science from The University of Texas at Dallas in 2005. He is currently working with the Digital Home Group at Intel Corporation in Portland, Oregon, USA.